Extracted from:

Swift Style

An Opinionated Guide to an Opinionated Language

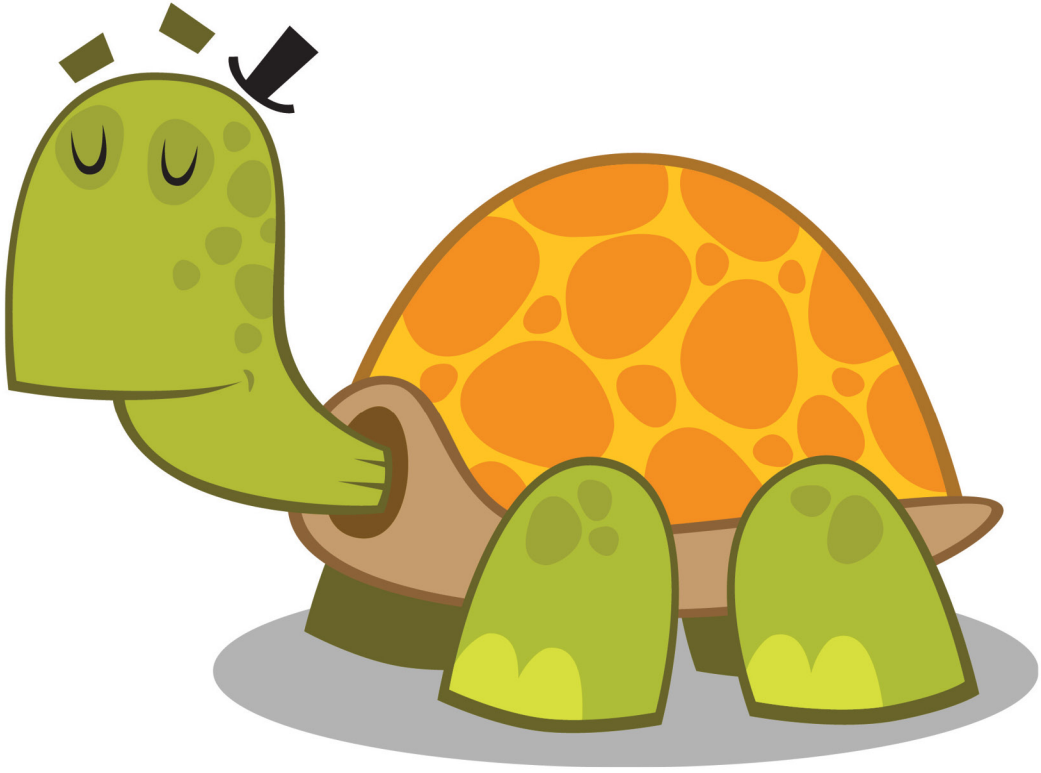
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Swift Style

An Opinionated Guide to an Opinionated Language



Erica Sadun

edited by Brian MacDonald

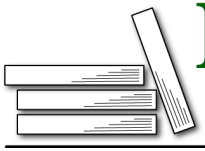
Swift Style

An Opinionated Guide to an Opinionated Language

Erica Sadun

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Susannah Davidson Pfalzer
Development Editor: Brian MacDonald
Indexing: Potomac Indexing, LLC
Copy Editor: Linda Recktenwald
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-235-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2017

Evaluating Case-Binding Syntax

Swift case binding is one of the least straightforward components of the entire language. Case binding enables you to match an enumeration case and then bind that case's associated values to new constants and variables. Where you place those let and var keywords to bind case values involves nontrivial choices.

Safety and consistency play important roles in these decisions. You must choose whether to favor readability or error prevention, even though the likelihood of errors is quite small.

This section explores two distinct styles—external and internal case binding—and details the advantages of each. I recommend you adopt a consistent internal style, which was not my practice prior to writing this book. In researching this topic, I learned that you can safely navigate a variety of pitfalls by preferring this less-attractive but more-reliable style.

External Case Binding

External case binding places a single let or var keyword outside the enumeration case. By moving keywords out of their tuples, you combine binding into a single additional keyword. The results use fewer characters and create a simpler syntax. Here are some examples that showcase both approaches:

```
enum StatusCode {
    case status(code: Int, message: String)
    case error(Error)
}

let fetch: StatusCode =
    .status(code: 418, message: "I'm a teapot")

if case .status(let code, let message) = fetch { // more words
    print(code, message) // 418 I'm a teapot
}

if case let .status(code, message) = fetch { // fewer words
    print(code, message) // 418 I'm a teapot
}
```

The external version also limits clutter. It produces a consistent code style for every binding, whether or not you ignore individual values. These next examples demonstrate the uniform presentation of prefixed let/var keywords. The style is uncomplicated and won't change based on the number of associated values or bound symbols:

```
enum Value<T> { case one(T), two(T, T), three(T, T, T) }
```

```
let example1: Value<Character> = .one("a")
let example2: Value<Character> = .two("a", "b")
let example3: Value<Character> = .three("a", "b", "c")

if case let .one(a) = example1 {}
if case let .two(a, _) = example2 {}
if case let .two(a, b) = example2 {}
if case let .two(_, b) = example2 {}
if case let .three(a, b, c) = example3 {}
if case let .three(_, b, c) = example3 {}
if case let .three(_, _, c) = example3 {}
```

Internal Case Binding

Internal case binding is not as pretty or consistent as its external alternative, but I've come around to adopting this style. The internal approach involves extra syntax for each bound symbol. Regardless, it is safer. Consistent internal binding, as in the following example, avoids errors introduced by a variety of uncommon edge cases:

```
if case .three(let a, let b, let c) = example3 {}
```

When pattern matching, it's common to bind a variable or constant and uncommon to use a bound value as an argument. Despite this rarity, adopting an "always explicit, always within the parentheses" rule adds consistency and safety to your code. The following example showcases an always-internal binding style used with an externally bound symbol. Under this style, binding is limited to each keyword's site. The oldValue constant will not be changed by the if-case statement:

```
let oldValue = "x"
...
// This safely binds and simultaneously matches.
if case .two(let newValue, oldValue) = example2 {
    ...
}
```

Consistent in-place binding avoids the accidental shadowing demonstrated in the following example. Overbinding shadow errors cannot happen when you adopt universal internal binding:

```
// This is an error because the intent is
// to bind newValue and match oldValue
if case let .two(newValue, oldValue) = example2 {
    // Wrongly matches "a", "b".
    //
    // `oldValue` is shadowed here, assigned the
    // value from the second field of the
    // enumeration's associated values.
```

```
    ... use newValue ...
}
```

Admittedly, this is an outlier case. Pattern matching rarely uses already-bound values. If you've adopted an external binding style, you can express this situation with a separate and explicit where or comma-delimited condition clause, as in the following example. This code introduces additional syntax and adds an extra variable binding (currentValue):

```
// This implements pattern binding and matching
// the given value but the extra condition separates
// these into two distinct goals
if case let .two(newValue, currentValue) = example2,
    currentValue == oldValue
{
    // correctly won't match "a", b"
    ... use newValue ...
}
```

Even here, safety can be problematic. If you inadvertently pass a wrong value to the condition clause, you'll introduce a hard-to-find error. In the following code, I've accidentally typed newValue when I meant to type oldValue. This code will compile but its logic is flawed. Consistent internal binding avoids this error, too.

```
// This error (`newValue` instead of `oldValue`) will not
// be caught by the compiler and is hard to catch by
// inspection.
if case let .two(newValue, currentValue) = example2,
    currentValue == newValue
{
    // correctly won't match "a", b"
    ... use newValue ...
}
```

Internal binding can be easier for new language adopters to read. Even without binding, if case is confusing. Both if case let and if case var (plus case var and case let) may look like single compound keywords rather than a combination of two distinct actions to developers unfamiliar with this syntax.

There's one final reason to adopt always-internal binding. When you need to mix let and var binding, you *must* use internal binding. This example shows how that might look in your code:

```
if case .three(_, let b, var c) = example3 {}
```

- Prefer consistent internal let and var binding. It's safe and simple.

- There is but *one let to rule them all and in the Swiftness bind them*. Use that one let internally and generously.

Ignoring Associated Values

Don't ignore the full complement of an enumeration case's associated values. Prefer to match cases instead. When you want to match only on the case, omit wildcard patterns and mention only the enumeration case. This allows you to ignore the content and structure of the payload and focus strictly on the enumerated conditions provided by the type:

```
switch statusCode {
case .error(_): // no
    ...
case .status(_, _): // extra no
    ...
}
```

```
switch statusCode {
case .error: // yes
    ...
case .status: // yes
    ...
}
```

Using If/Guard-Case

Both if-case and guard-case are most valuable when used to bind associated values to variables. They allow you to reach inside an enumeration and access those values based on an enumeration's specific cases:

```
enum Result<T> { case success(T), error(Error) }
guard case .success(let value) = result else { return } // yes
print(value)
```

When pattern matching, prefer the ~= operator to if-case and guard-case. The following statements perform identical tests, checking whether myValue falls within a range:

```
if range ~= myValue { print("success") } // yes
if case range = myValue { print("success") } // no
```

Of these, the first is readable and simple. The second, while logically identical, is esoteric and hard to process. Using the equal sign when there's no assignment or condition binding promotes confusion for most Swift developers, even those experienced with pattern-matching syntax.

While coders can be trained to recognize this rare use, you cannot be sure that anyone reading your code will have that training. This rule falls under the mantle of Brian Kernighan's admonition against writing clever code.[13] Prefer the obvious to the obscure, no matter how cool you consider the alternative to be.

- Reserve `if-case` and `guard-case` to bind variables.
- Prefer `~=` for non-binding pattern matching.
- "Too clever is dumb."—Ogden Nash

Choosing Capture Modifiers

Prefer `weak` capture to `unowned`. Weak captures can be checked at the start of a closure and, if still valid, assigned to a strong reference for the life of the enclosing scope. They provide excellent safety and utility.

13. https://en.wikiquote.org/wiki/Brian_Kernighan

An unowned capture is equivalent to Objective-C's unsafe-unretained; it can be used when "the closure and the instance it captures will always refer to each other, and will always be deallocated at the same time." You may use unowned items to refer to global instances whose lifetime extends throughout the application's duration.

An unowned reference guarantees that a reference *always* has a value. If there is any chance whatsoever that the unowned item may deallocate, using unowned items is roughly equivalent to and as desirable as using a weak value with forced unwrapping. The advantage—if that's the right word to use—is that an unowned reference is not an optional value and can be used without further unwrapping.

When capturing weak self in a closure, prefer to create a strong reference at the start of the closure rather than prefix all calls with self?.member optional chaining. Introducing an early strong reference ensures that self cannot deallocate throughout the lifetime of the closure's scope.

```
[weak self] in
guard let strongSelf = self // yes
    else { ... leave scope ... }
strongSelf.f()
strongSelf.g()
```

```
[weak self] in
self?.f() // no
self?.g()
self?.h()
```

- weak capture is generally safer than unowned capture.
- If you must use unowned, reserve unowned capture for items with guaranteed lifetimes.
- While singletons can be seen as good candidates for unowned capture, since their existence is guaranteed throughout the lifetime of the program, there are no real worries about deallocation. A strong reference may be simpler to use.
- Capturing a weak version of self ensures you won't create reference cycles in your code that prevent memory from properly deallocating.
- Some developers prefer to use this in preference to strongSelf. (I feel that strongSelf is better at self-documenting.) The arguments in favor of this are similarity in length (this is the same number of letters as self) and it has a historic meaning similar to self in other languages.

Other Practices

Here are some handy ways to help optimize and improve your code:

- Prefer let constants to var variables unless you truly need mutable data. The compiler can better optimize your data when it's guaranteed not to change. The Swift compiler is very good at detecting unmodified var use.
- Marking classes as final enables the compiler to introduce performance improvements. These improvements depend on moving away from dynamic dispatch, which requires a runtime decision to select which implementations to call. Removing indirect calls for methods and property access greatly improves your code performance.
- Swift's whole-module optimization can automatically infer many final declarations by scanning and compiling an entire module at once. Its inference capabilities apply only to constructs and members marked as internal, fileprivate, and private. Class members with public access must explicitly declare final to participate in this optimization.
- Avoid escaping closure arguments unless they must outlive your functions. Using nonescaping closures (the default) introduces performance optimizations and bypasses the need to annotate properties and methods with self.
- Default arguments involve a minor check with a small overhead cost. Unless you plan to run tens of millions of calls at once, don't let this overhead sway you away from providing defaults. Using default values, even for closure arguments, involves minimal overhead and significant usability gains.
- Avoid complex string interpolation that performs too much work within the \() interpolation delimiter. This is best broken out into separate statements to improve readability and maintainability.

Appending

Prefer append() and append(contentsOf:) to using the + operator when concatenating elements and arrays in situations where performance matters. This enables the Swift compiler to better optimize your code:

```
var results: [Iterator.Element] = [] // yes
results.reserveCapacity(rest.count + 2)
results.append(self[first]); results.append(self[second])
results.append(contentsOf: rest.lazy.map({ self[$0] }))
```

```
return [self[idx1], self[idx2]] // non-performant
    + rest.lazy.map({ self[$0] })
```

There's no reason to avoid += since it calls append(contentsOf:) in its implementation.

Counting

Don't test a collection's count property. Prefer checking isEmpty over count == 0. The isEmpty property returns a Boolean value indicating whether the collection contains no elements. It operates in O(1) time in most (but not all) cases. The standard library recommends using isEmpty over testing count, especially for computed and potentially infinite sequences, which is where count can be particularly expensive.

isEmpty is not always O(1). A lazy filter with no matches must run through its entire sequence before determining isEmpty (which is O(N) complexity). isEmpty should be O(1) for all non-lazy collections and for many lazy collections too, such as lazy maps.